LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-JC-150863

Out-of-Core Construction and Visualization of Multiresolution Surfaces

P. Lindstrom

November 4, 2002

Association for Computing Machinery Siggraph 2003 Symposium on Interactive 3D Graphics, Monterey, CA, April 27-30, 2003

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Out-of-Core Construction and Visualization of Multiresolution Surfaces

Peter Lindstrom

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Abstract

We present a method for end-to-end out-of-core simplification and view-dependent visualization of large surfaces. The method consists of three phases: (1) memory insensitive simplification; (2) memory insensitive construction of a multiresolution hierarchy; and (3) run-time, output-sensitive, view-dependent rendering and navigation of the mesh. The first two off-line phases are performed entirely on disk, and use only a small, constant amount of memory, whereas the run-time system pages in only the rendered parts of the mesh in a cache coherent manner. As a result, we are able to process and visualize arbitrarily large meshes given a sufficient amount of disk space; a constant multiple of the size of the input mesh.

Similar to recent work on out-of-core simplification, our memory insensitive method uses vertex clustering on a uniform octree grid to coarsen a mesh and create a hierarchy, and a quadric error metric to choose vertex positions at all levels of resolution. We show how the quadric information can be used to concisely represent vertex position, surface normal, error, and curvature information for anisotropic view-dependent coarsening and silhouette preservation.

The run-time component of our system uses asynchronous rendering and view-dependent refinement driven by screen-space error and visibility. The system exploits frame-to-frame coherence and has been designed to allow preemptive refinement at the granularity of individual vertices to support refinement on a time budget.

Our results indicate a significant improvement in processing speed over previous methods for out-of-core multiresolution surface construction. Meanwhile, all phases of the method are both disk and memory efficient, and are fairly straightforward to implement.

1 INTRODUCTION

Recent advances in scanning technology and the ever increasing size of computer simulations have lead to a rapid increase in the availability and size of geometric data sets. Massive polygonal data sets, consisting of hundreds of millions of faces, are becoming quite common [2, 19]. While the performance of graphics hardware has also seen a dramatic rise in the last few years, our ability to produce data sets that overload the capabilities of state-of-the-art graphics chips has lead researchers to develop methods for automatic model simplification and run-time level-of-detail (LOD) management [12]. Whereas many polygonal environments consist of a large collection of moderately complex objects, such as those used in video games, recent trends are for single objects to consist of millions of polygons. Examples of such models include terrain surfaces and high-resolution range scans. The traditional approach of storing a few static levels of detail is not viable for such large objects, which are often viewed in a manner that they vary greatly in screen resolution over the surface. As a result, methods for view-dependent simplification have been proposed, in which a continuous level-of-detail hierarchy is first constructed, and is then adapted at a fine granularity at run-time [7, 9, 10, 17, 22, 25, 33].

Contact info: 7000 East Avenue, L-560 · Livermore, CA 94551 · pl@llnl.gov.
This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48.

The techniques mentioned above have been used successfully for simplifying models up to a few million triangles. In the last few years, however, polygonal models have become so large that they often greatly exceed our ability to perform conventional in-core simplification, and a number of techniques have been devised for simplifying such large models out-of-core, e.g. [2, 21, 24, 30]. These methods, however, all produce static, single-resolution meshes, which in a sense is inappropriate for such large surfaces, because in order to view them interactively they have to be simplified to such a degree that many important details are lost. Rather, we would like to construct a level-of-detail hierarchy for such large surfaces, and then use view-dependent techniques to render and explore them at varying resolution without significant loss in fidelity. While there have been some recent publications on out-of-core construction of LOD hierarchies, most notably the work by El-Sana and Chiang [8] and by Prince [27], these methods have rather long execution times, can be somewhat difficult to implement, and still rely on having a large amount of memory available for in-core processing. Furthermore, the results presented in these papers are for models of rather modest size—just a million or a few million triangles, making it difficult to judge how well they scale to truly large models.

In this paper, we present an alternative approach to out-of-core simplification for view-dependent refinement, by extending the static simplification algorithm by Lindstrom and Silva [24]. Our end-to-end off-line method is *memory insensitive*, meaning that it can run successfully with essentially an arbitrarily small amount of memory (in our case, less than 8 MB). We achieve this by storing all intermediate computations in temporary files on disk, and take advantage of fast sequential disk access. In addition to being memory efficient, our method is considerably faster than previously published techniques, running at a triangle reduction rate of up to 60,000 triangles per second. We also present data structures for concisely encoding *quadric matrices* and the per-vertex information needed later at run-time during view-dependent rendering. Finally, our method is straightforward to implement and is a simple but significant extension of Lindstrom and Silva's memory insensitive simplification algorithm. We will present the various steps in our algorithm after covering related work in the area.

2 PREVIOUS WORK

In this section, we cover previous work on view-dependent and out-of-core surface simplification. Because there has been extensive work on simplification, we here only cover a few of the most related algorithms in the field.

Among the first methods for view-dependent simplification for general polygonal models was the technique by Xia and Varshney [33]. Their method uses *edge collapse* to construct a binary tree of possible coarsening operations off-line. At run-time, they use a screen space metric, based on geometric error and proximity to silhouettes and specular highlights, for determining which edges to collapse. Hoppe extended his work on progressive meshes to view-dependent refinement [17]. Similar to [33], Hoppe uses edge collapse, but his algorithm provides greater freedom in choosing the order of edge collapses, which generally results in higher quality adaptive meshes. His method also makes use of geomorphing to

reduce temporal LOD artifacts. More recently, El-Sana and Varshney [9] presented a method, based on the more general vertex merge operation, that is able to merge and simplify topologically disjoint parts of an object.

A different approach to view-dependent refinement was taken by Luebke and Erikson [25]. Rather than relying on edge collapse or vertex pair contraction, they use an even more general coarsening operation—vertex clustering—which allows a large collection of vertices to be merged in a single atomic operation. Like us, they make use of an octree decomposition of space, rather than a general binary tree over the set of mesh vertices. Like others, they make use of normal cones to detect when the surface is near a silhouette. Luebke and Hallen [26] later used this framework for performing view-dependent "imperceptible simplification."

The off-line processing techniques above all assume that the hierarchy can be constructed in-core. To simplify large meshes out-of-core, Lindstrom [21] proposed a technique, based on vertex clustering on a uniform grid, that makes use of Garland and Heckbert's *quadric error metric* [14]. Lindstrom's method performs a single sweep over the mesh and constructs an in-core representation of the simplified model. More recently Lindstrom and Silva [24] extended this method by removing the requirement of having enough RAM to store the simplified model. Their memory insensitive technique uses a constant amount of memory and performs a series of external sorts to allow sequential access to the on-disk data. Our first simplification phase is based upon their algorithm. To provide a higher level of adaptivity, Shaffer and Garland [30] proposed making two instead of one pass over the mesh. In the first pass, uniform clustering like in [21] is performed, after which a binary space partitioning (BSP) tree is constructed from the accumulated quadric information. In the second pass, the BSP tree is used to recluster the mesh. Other recent out-of-core simplification methods for adaptive but static levels of detail are presented in [3, 11, 15].

There have only been a few published methods for out-of-core simplification for view-dependent refinement. Hoppe applied his view-dependent progressive mesh work to terrain data [18]. His approach is to partition the terrain into a block hierarchy and simplify the blocks independently. Then, the blocks are merged and the seams between them are simplified further. Prince [27] later extended Hoppe's out-of-core simplification method for terrain to arbitrary polygonal surfaces. Prince's method, like ours, makes use of quadric error metrics, but uses edge collapse as the coarsening primitive. While effective for medium-size models, his out-of-core method still requires much RAM and may be too slow for simplifying very large models. Cignoni et al. [4] describe efficient data structures for a similar block hierarchy based technique that allows a greater degree of simplification across block boundaries. Their results suggest an improvement over [27] in simplification speed and memory efficiency, but they do not report on using their data structures for interactive view-dependent refinement. El-Sana and Chiang [8] proposed a novel out-of-core technique for view-dependent simplification by segmenting the mesh into independent patches. These patches are such that the edge collapse order inherently preserves the boundaries between them, thus simplifying and stitching the patches together can be done without the need for explicit boundary constraints. Unfortunately, the models used by El-Sana and Chiang are small by out-of-core simplification standards, and it is not clear how well their method scales. For large enough models the average patch size is likely to shrink to the point that each edge must be loaded and collapsed individually, leading to excessive thrashing. More recently, DeCoro and Pajarola [6] described external data structures and a run-time system for out-of-core visualization. Their framework assumes edge collapse is used to coarsen the mesh, but no out-of-core method is presented for constructing the multiresolution structure.

Rusinkiewicz and Levoy [29] proposed an interesting alternative to polygon-based view-dependent refinement. Their *QSplat* algorithm clusters the triangle mesh into a vertex hierarchy, and then uses point primitives to render the mesh. While conceptually simple, most current hardware is optimized for triangle rather than point rendering, and the quality afforded by real-time point-based rendering can be rather low. Still, hybrid techniques like Cohen et al.'s point- and triangle-based simplification [5] may prove useful. Finally, there are a number of out-of-core visualization systems based on managing collections of static levels-of-detail, including [1, 10, 32]. While related to our method, the focus of these works are on the run-time handling of a large number of small objects, as opposed to the off-line construction and run-time fine-grained adaptation of a single large mesh.

To our knowledge, our system is the first to perform both scalable out-of-core simplification and view-dependent visualization of general surfaces from external memory. We will describe our algorithm in the following sections.

3 ALGORITHM OVERVIEW

Our view-dependent algorithm consists of three phases: simplification, level-of-detail hierarchy construction, and run-time view-dependent refinement and rendering. The first two phases are run off-line, and are used to produce an on-disk level-of-detail representation of the mesh. The run-time component then traverses this hierarchy, pages in the data needed, and produces an adaptive mesh that can be displayed interactively. Our main approach is to use a sparse octree decomposition of space over a uniform rectilinear grid, similar to Luebke and Erikson's view-dependent simplification algorithm [25]. The octree is sparse in the sense that only those nodes that contain at least one vertex from the input mesh are retained. Each node in this octree corresponds to a vertex at some level of resolution, and adaptive mesh simplification and refinement are performed by collapsing and expanding nodes (i.e. removing and creating child nodes, respectively) in the octree. In this section, we give a brief overview of the three phases of our algorithm, and provide further details in the following sections.

The first phase—simplification—is based on the *OoCSx* memory insensitive mesh simplification algorithm by Lindstrom and Silva [24]. Their method is a memory efficient variation on the out-of-core simplification algorithm by Lindstrom [21], which in turn was inspired by Rossignac and Borrel's [28] vertex clustering algorithm. Using a uniform grid to partition space, all vertices that fall in the same grid cell are merged (clustered) to a single vertex. In this process, the triangles that collapse to an edge or a point are discarded. Representative vertices for the clusters are chosen based on minimizing the *quadric error*—a weighted sum of squared distances to the triangle planes of the input mesh—which can be encoded using a symmetric 4×4 matrix [14].

The OoCSx algorithm performs all these tasks on disk, and avoids costly random accesses using a sequence of external sorts, followed by fast sequential accesses. The intermediate output of this algorithm is a set of triangles for the simplified mesh and a list of quadric matrices for its vertices. Each triangle that survives the simplification is later assigned to an internal node in the octree hierarchy, as given by an *octcode*—a bit string that uniquely identifies a grid cell by position and resolution in the octree. Also to facilitate the ensuing octree construction, we augment the per-vertex quadric matrices with a vertex position, computed from the quadric matrix, and an approximate vertex normal, and output these fields to a vertex file. The vertex file is output in octcode order, such that sibling nodes are stored together. This phase of the algorithm is described in Section 4.

The octree construction phase begins by sorting the triangle file by octcode, such that the triangle order follows the order in which nodes are created in the octree. We then build the octree by first processing the vertex file sequentially. Sibling nodes are identified

simplify(T_{in})

for each triangle $t = \langle \mathbf{p}_t^1, \mathbf{p}_t^2, \mathbf{p}_t^3 \rangle \in T_{in}$
  compute plane $\bar{\mathbf{n}}_t$ for t
  for each vertex $\mathbf{p}_t^i$ of t
    map $\mathbf{p}_t^i$ to leaf octcode v_t^i
    append $\langle v_t^i, \bar{\mathbf{n}}_t \rangle$ to plane file P
  if v_t^1, v_t^2, v_t^3 are distinct **then**
    compute octcode v_t for t from v_t^1, v_t^2, v_t^3
    append $\langle v_t^*, v_t^1, v_t^2, v_t^3 \rangle$ to triangle file T_{out}
externally sort P on octcode v
for each octcode $v \in P$
  for each plane $\bar{\mathbf{n}}_t$ for v
    add $\bar{\mathbf{n}}_t \bar{\mathbf{n}}_t^\mathsf{T}$ to quadric matrix $\mathbf{Q}_v$ for v
    add $\hat{\mathbf{n}}_t$ to approximate normal $\hat{\mathbf{n}}_v$ for v
  compute vertex position $\mathbf{p}_v$ and error ϵ_v from $\mathbf{Q}_v$
  append $\langle v, \mathbf{Q}_v, \mathbf{p}_v, \hat{\mathbf{n}}_v \rangle$ to vertex file V_n

Table 1: Pseudo-code for memory insensitive simplification. The output is a triangle file T_{out} and a vertex file V_n for level n (the bottom level) in the LOD hierarchy.

and grouped together, and the quadric and other geometric information is merged to form a new level. Meanwhile, the triangle file is scanned sequentially and triangles are deposited in the nodes where they belong. The generated parent nodes are output sequentially (again in octcode order) to a temporary file for that given level of resolution, and the process is applied iteratively until all levels in the octree are represented, ending with a temporary file containing a single node—the root node. This bottom-up construction is then followed by a final top-down, level-by-level traversal, during which nodes are linked together in a single file and the hierarchical structure is output in the desired order, from coarse to fine resolution. Note that all the I/O accesses used in our simplification and octree construction are sequential in nature, and are therefore very efficient. The algorithm for the octree construction phase is described in Section 5, while the data structures for the output produced are discussed in detail in Section 6.

In the run-time phase, the file containing the LOD hierarchy, which could potentially greatly exceed the available main memory, is memory mapped so that it can be accessed as though it were resident in contiguous memory. For simplicity, we let the operating system perform on-demand paging of the external data, although this data could equally well be accessed using an explicit paging scheme. To minimize run-time computation, an in-core "copy" is made of each active node in the octree. As new nodes are created, we extract data from the external LOD hierarchy, compute any per-vertex and per-triangle information not explicitly stored, and write this data to a dynamically allocated data structure. These nodes are maintained in a dynamic subset of the complete external octree, and are expanded and collapsed, as determined by view-dependent error and visibility criteria, to adapt to the current viewpoint. The triangles contained in the active nodes and the currently active vertices they reference are then output to a triangle list and rendered. To avoid stalling the rendering when parts of the mesh need to be paged in, we decouple renderering and refinement/paging as two asynchronous threads. Also, for the sake of interactivity, we impose a time limit on the refinement thread to guarantee frequent updates of the mesh, and generally perform refinement breadth-first to ensure that detail is progressively added evenly over the mesh whenever the allotted refinement time is insufficient. The in-core, dynamic data structures are presented in Section 7, while the steps of the view-dependent refinement are given in Section 8.

4 SIMPLIFICATION

The simplification phase of our algorithm is based on, and is essentially identical to, the beginning stages of the memory insensitive technique *OoCSx* described in [24]. Therefore, we will only briefly cover this part of our algorithm, and we will focus on the few differences between the two methods. Pseudo-code for the simplification phase is given in Table 1.

Before the simplification begins, the user chooses the resolution of a uniform rectilinear grid that completely contains the input mesh. This grid is constrained to have dimensions $2^n \times 2^n \times 2^n$, for some positive integer n. The cells in this grid correspond to the leaf nodes in an $(n+1)$-level octree that ultimately forms a multiresolution representation of the mesh. In the discussion below, we will use the terms grid cell, cluster, and node interchangedly.

As in [24], we process the input mesh, which is represented as a triangle soup (a sequence of triplets of vertex coordinates), one triangle at a time. For each triangle t, we compute a 4-vector

$$\bar{\mathbf{n}}_t = w_t \begin{pmatrix} \hat{\mathbf{n}}_t \\ d_t \end{pmatrix} \tag{1}$$

for an implicit plane equation $\hat{\mathbf{n}}_t^\mathsf{T}\mathbf{x} + d_t = 0$. In our implementation, the weight w_t is set to the area A_t of t. Then, for each of t's three vertices, we quantize the vertex coordinates to an integer grid cell location, and then convert this location to an *octcode* v. These octcodes are represented as follows: The root node has octcode $v = 1$ and the k^{th} child of a node v is computed as $8v + k$, with $k = 0, \ldots, 7$. These octcodes have the property that they are ordered by level, from top to bottom, and sibling nodes have consecutive octcodes. Whereas these octcodes are different from the cluster IDs used in OoCSx, this is of no consequence to the simplification algorithm—any one-to-one mapping between quantized coordinates and cluster IDs will do. The planes and associated octcodes are then output sequentially to a temporary plane file P. If the triangle's vertices belong to different clusters, we determine which octree node v_t to assign t to. As in [25], we choose v_t to be the lowest common ancestor of all pairs of vertices from t, such that expanding the node v_t would introduce the triangle t (analogous to the introduction of triangles caused by a vertex split in [16]). Rather than storing v_t directly with t, we assign to t a sort key v_t^* that also orders the nodes by increasing octcode within each level, but orders the levels from bottom to top instead of from top to bottom.[1] This is the order in which we will later begin constructing the octree. Finally, the non-degenerate triangle and its reversed octcode are written to a triangle file T_{out}.

After the input has been exhausted, the plane file is sorted on the octcode field using an external sort. As in [24], we use `rsort` [20] for this task. We then process all planes, one cluster at a time, and construct a 4×4 quadric matrix $\mathbf{Q}_v$ for each cluster v:

$$\mathbf{Q}_v = \sum_t \bar{\mathbf{n}}_t \bar{\mathbf{n}}_t^\mathsf{T} \tag{2}$$

We also construct an approximate normal for the cluster, which will later be used to resolve an ambiguity in sign of the actual normal, extracted from the quadric matrix. If space is at a premium, we can avoid storing $\hat{\mathbf{n}}_v$ by performing additional computation and encoding this sign directly in the quadric matrix (see Section 6.1.2). Given a quadric matrix $\mathbf{Q}_v$, we proceed by computing a representative vertex for the cluster that minimizes the quadric error [24]. Finally, we output the quadric matrices, along with their octcodes, vertex positions, and normals, to a vertex file V_n for level n in the octree (the bottommost level), and we remove the temporary plane file P. For each leaf node that contains at least one vertex from the original mesh, we now have a single vertex for the full-resolution simplified mesh. The original OoCSx simplification algorithm would at this point perform multiple external sorts on the triangle file to replace the cluster IDs with vertex indices. Because

[1] This key can be computed from the octcode by inverting all bits lower than the most significant one bit and negating the result.

octree-construct(V_n, T)

```
 1 externally sort T on reversed octcode v*
 2 for each level l = n, ..., 1
 3     for each set of siblings C in V_l with parent octcode p
 4         compute Q_p and n̂_p by summing over children
 5         compute p_p, ε_p, and r_p
 6         record offsets O_p of p's children within V_l
 7         for each triangle ⟨v*_t, v^1_t, v^2_t, v^3_t⟩ ∈ T with v_t = p
 8             append ⟨v^1_t, v^2_t, v^3_t⟩ to T_p
 9         append ⟨p, Q_p, p_p, n̂_p, ε_p, r_p, O_p, T_p⟩ to V_{l-1}
10 for each level l = 0, ..., n
11     for each node p in V_l
12         if l = n then
13             append ⟨p_p, n̂_p⟩ to octree hierarchy H
14         else
15             compute r_p and λ_p from Q_p and n̂_p
16             compute child offsets C_p within H from O_p
17             append ⟨r_p, λ_p, p_p, ε_p, r_p, C_p, T_p⟩ to H
```

Table 2: Pseudo-code for memory insensitive octree construction. The procedure takes as input a vertex file V_n and a triangle file T, and outputs a LOD hierarchy H.

we will make direct use of the cluster IDs (or octcodes), this step fortunately does not have to be done in our algorithm. Instead, we perform only a single external sort—on the plane equation file.

The steps described above are all the components of our simplification algorithm. The output is a single-resolution, static mesh. We now proceed by constructing a level-of-detail hierarchy for this simplified mesh.

5 LOD HIERARCHY CONSTRUCTION

The second phase of our algorithm takes the simplified mesh, represented as a list of vertices V_n, augmented with quadric error information, and a list of triangles T, and constructs a coarse-to-fine level-of-detail representation H of the mesh. For each node in H, we store vertex information, such as position, normal, error, etc., as well as a (potentially empty) list of triangles. The triangles of a node are the ones that are eliminated when the node is collapsed. We will focus later on the particular data structures used for the nodes in H, and spend this section describing the steps of the LOD construction algorithm.

Table 2 lists pseudo-code for the octree construction. We begin by externally sorting the triangle file on its reversed octcode field v^*. This is done to group the triangles by octree node and to place them in the order in which the nodes will be processed during octree construction. Following this sort, we then begin constructing the internal nodes of the octree (lines 2–9). Recall that the simplification has already produced vertex positions and normals for the leaf nodes V_n on level n. Because V_n is sorted on octcode, and because sibling nodes have consecutive octcodes, we can easily scan V_n and fetch the per-vertex data for each group of siblings. From these we compute a quadric matrix and approximate normal for the parent using simple addition. We additionally compute the optimal vertex position $\mathbf{p}$, the minimum error ϵ, and the radius r of a bounding sphere, centered on $\mathbf{p}$, that encloses the bounding spheres of the node's children (the bounding spheres for leaf nodes have zero radius). As child nodes are read from V_l, we record their file offsets within V_l and store these with their parent. These offsets will later be used to construct links between nodes in the hierarchy. After the vertex data and offsets have been computed, we sequentially scan the triangle file for those triangles assigned to the parent node. Finally, we append the parent node to a temporary file V_{l-1} for the next coarser level. This procedure is iterated until nodes for all levels of the octree have been constructed, after which T has been exhausted and can safely be removed.

The next and final step is to link the nodes together in a single file H (lines 10–17). As is common in multiresolution methods, we store the multiresolution structure from coarse to fine resolution. While this layout is of no particular advantage to our view-dependent algorithm, other than the fact that the breadth-first layout and closeness of siblings in the file result in good cache coherence, we anticipate that a coarse-to-fine order would be beneficial for progressive transmission.

Starting with the root node and processing one level at a time, we compute for each internal node p an alternative representation $(\mathbf{r}_p, \boldsymbol{\lambda}_p)$ for $\mathbf{Q}_p$, which also implicitly encodes the normal $\hat{\mathbf{n}}_p$. We will describe this representation in more detail in Section 6. For the internal nodes, we also compute global offsets C_p to their children within H. Using the offset o_l to the beginning of the current level l, the byte size $|V_l|$ of the current level (as recorded at the end of outputting V_l during the bottom-up construction), and the offset o_c to one of p's existing children within the next level $l+1$, the position of c in H is simply $o_{l+1} + o_c = o_l + |V_l| + o_c$. We thus replace O_p with C_p and append p to H. For leaf nodes, we write only the fields $\langle \mathbf{p}, \hat{\mathbf{n}} \rangle$. After the temporary file V_l has been exhausted, we can remove it and move on to the next level. Finally, after all nodes have been output to H, phase 2 of our algorithm is complete.

on-disk non-leaf node		
vertex data		
3-vector	$\mathbf{r}$	rotation for orthogonal matrix $\mathbf{P}$
3-vector	$\boldsymbol{\lambda}$	eigenvalues of $\mathbf{A}$
3-vector	$\mathbf{p}$	vertex position
scalar	ϵ	quadric error at $\mathbf{p}$
scalar	r	bounding sphere radius
triangle data		
count	n_T	number of triangles
octcode*3	T	list of triangles
octree data		
offset*8	C	file offsets to children
on-disk leaf node		
vertex data		
3-vector	$\mathbf{p}$	vertex position
3-vector	$\hat{\mathbf{n}}$	normal vector

Table 3: On-disk data structures for octree non-leaf and leaf nodes.

6 EXTERNAL DATA STRUCTURES

We now turn our attention to the data structures used for our external on-disk representation of the level-of-detail hierarchy H. Each node in the hierarchy consists of vertex information (position, normal, etc.) and, if the node is not a leaf, a list of triangles and pointers to its children. These data structures are given by Table 3. Because leaf nodes have no triangles associated with them, and because error evaluation and view culling are done only for internal nodes, we store only position and normal information with leaf nodes. The triangles are represented as triplets of octcodes corresponding to leaf nodes in the hierarchy. The computation of the vertex data from the quadric matrices is more involved, and we will describe the on-disk vertex fields in the following sections.

6.1 Vertex Data

In this section, we describe how to compute and store the per-vertex information needed in our view-dependent renderer from the 4×4 quadric matrix $\mathbf{Q}$. The data we are primarily concerned with are the vertex position $\mathbf{p}$, the surface normal $\hat{\mathbf{n}}$, the quadric error ϵ at $\mathbf{p}$, and a matrix $\mathbf{K}$ that encodes the normal curvature and is used to measure how large the error appears from different view directions. We could compute and store $(\mathbf{K}, \hat{\mathbf{n}}, \mathbf{p}, \epsilon)$ directly, however this information would require $6 + 3 + 3 + 1 = 13$ scalar values, whereas the original quadric matrix requires only 10 values (assuming we take advantage of the fact that $\mathbf{Q}$ and $\mathbf{K}$ are symmetric). Instead, we will make use of an alternative representation

$(\mathbf{r}, \boldsymbol{\lambda}, \mathbf{p}, \epsilon)$—three 3-vectors and a scalar—that allows $\mathbf{K}$ and $\hat{\mathbf{n}}$ to be computed quickly. In addition to these fields, we also store the radius r of a bounding sphere, centered on $\mathbf{p}$, used for view frustum culling. We will describe how to compute this radius later.

6.1.1 Vertex Position and Quadric Error

In Section 4, we explained how to compute the quadric matrix $\mathbf{Q}$ for a vertex or, more generally, a node in the LOD hierarchy. As in [21], we decompose the quadric matrix as

$$\mathbf{Q} = \begin{pmatrix} \mathbf{A} & -\mathbf{b} \\ -\mathbf{b}^\mathsf{T} & c \end{pmatrix} \tag{3}$$

We can then write the quadric error $\epsilon(\mathbf{x})$ as

$$\epsilon(\mathbf{x}) = \mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} - 2\mathbf{b}^\mathsf{T}\mathbf{x} + c \tag{4}$$

Let $\mathbf{p}$ be the position of the vertex associated with the node and let $\nabla\epsilon(\mathbf{p}) = 2(\mathbf{A}\mathbf{p} - \mathbf{b})$ be the gradient of ϵ at $\mathbf{p}$. We now rewrite ϵ:

$$\begin{aligned} \epsilon(\mathbf{x}) &= (\mathbf{x}-\mathbf{p})^\mathsf{T}\mathbf{A}(\mathbf{x}-\mathbf{p}) + 2(\mathbf{A}\mathbf{p}-\mathbf{b})^\mathsf{T}(\mathbf{x}-\mathbf{p}) \\ &\quad + (\mathbf{p}^\mathsf{T}\mathbf{A}\mathbf{p} - 2\mathbf{b}^\mathsf{T}\mathbf{p} + c) \\ &= (\mathbf{x}-\mathbf{p})^\mathsf{T}\mathbf{A}(\mathbf{x}-\mathbf{p}) + \nabla\epsilon(\mathbf{p})^\mathsf{T}(\mathbf{x}-\mathbf{p}) + \epsilon(\mathbf{p}) \end{aligned} \tag{5}$$

Because we want to minimize the quadric error, we generally choose the vertex position $\mathbf{p}$ to be a minimizer for ϵ, which implies $\nabla\epsilon(\mathbf{p}) = \mathbf{0}$ and

$$\begin{aligned} \epsilon(\mathbf{x}) &= (\mathbf{x}-\mathbf{p})^\mathsf{T}\mathbf{A}(\mathbf{x}-\mathbf{p}) + \epsilon(\mathbf{p}) \\ \epsilon(\mathbf{p}) &= c - \mathbf{p}^\mathsf{T}\mathbf{A}\mathbf{p} \end{aligned} \tag{6}$$

That is, if $\mathbf{p} = \operatorname{argmin}\epsilon$, we can parameterize the quadric error as $(\mathbf{A}, \mathbf{p}, \epsilon)$. This parameterization is valid whether $\mathbf{A}$ is singular or not. If $\mathbf{A}$ is singular, ϵ has infinitely many minima, in which case we choose the one closest to the grid cell center. In rare circumstances our chosen optimum $\mathbf{p}$ falls outside its associated grid cell. When this happens, we constrain $\mathbf{p}$ using the procedure outlined in [24], which may result in a $\mathbf{p}$ that does not minimize ϵ. Regardless of this special case, if we do not have to compute ϵ at points other than the vertex position $\mathbf{p}$ (irrespective of our choice of $\mathbf{p}$), then the parameterization $(\mathbf{A}, \mathbf{p}, \epsilon)$ is useful since it directly gives us the vertex position $\mathbf{p}$ and the error ϵ at $\mathbf{p}$. Still, we are left with determining the normal $\hat{\mathbf{n}}$ and the curvature matrix $\mathbf{K}$. As shown below, these two quantities can both be derived from the matrix $\mathbf{A}$.

6.1.2 Surface Normal

To compute the surface normal $\hat{\mathbf{n}}$, note that the matrix $\mathbf{A}$ is the covariance matrix (with zero mean) for the set of (weighted) normals of the triangles in the cluster [13]. Thus, the eigenvector for the largest eigenvalue λ_1 of $\mathbf{A}$ corresponds to the dominant normal direction $\hat{\mathbf{n}}$. Note that if $\hat{\mathbf{n}}$ is an eigenvector, then so is $-\hat{\mathbf{n}}$. Because the sign of the normal matters for correct rendering, we will show later how to resolve this ambiguity. Using an eigen decomposition of $\mathbf{A}$, we have

$$\mathbf{A} = \mathbf{P}\boldsymbol{\Lambda}\mathbf{P}^\mathsf{T} \tag{7}$$

$$\mathbf{P} = (\hat{\mathbf{x}}_1 \quad \hat{\mathbf{x}}_2 \quad \hat{\mathbf{x}}_3) \quad \hat{\mathbf{x}}_1 \parallel \hat{\mathbf{n}} \tag{8}$$

$$\boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \lambda_2, \lambda_3) \quad \lambda_1 \geq \lambda_2 \geq \lambda_3 \geq 0 \tag{9}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix of (non-negative) eigenvalues, and $\mathbf{P}$ is orthogonal with determinant $\det(\mathbf{P}) = 1$. That is, $\mathbf{P}$ is a rotation matrix, which can be represented using as little as three parameters. We have chosen to use a 3-parameter axis-angle representation that is similar to the standard unit quaternion representation. Let $\mathbf{P}$ correspond to a rotation around a unit vector $\hat{\mathbf{r}}$ by an angle θ. Then the vector

$$\mathbf{r} = (r_x \quad r_y \quad r_z)^\mathsf{T} = \sqrt{2}\sin\tfrac{\theta}{2}\,\hat{\mathbf{r}} \tag{10}$$

completely represents $\mathbf{P}$. We will not go into detail of how to compute $\mathbf{r}$ from $\mathbf{P}$, but refer the reader to any tutorial on quaternions, e.g. [31]. We recover $\mathbf{P}$ from $\mathbf{r}$ as follows:

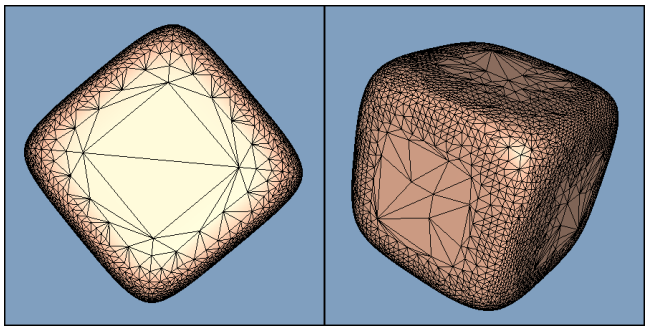
$$\mathbf{P} = \begin{pmatrix} 1-r_y^2-r_z^2 & r_xr_y - \alpha r_z & r_xr_z + \alpha r_y \\ r_yr_x + \alpha r_z & 1-r_z^2-r_x^2 & r_yr_z - \alpha r_x \\ r_zr_x - \alpha r_y & r_zr_y + \alpha r_x & 1-r_x^2-r_y^2 \end{pmatrix} \tag{11}$$

$$\alpha = \sqrt{2 - r_x^2 - r_y^2 - r_z^2} \tag{12}$$

Thus, by storing $\mathbf{r}$, we can quickly compute $\mathbf{P}$ and the normal $\hat{\mathbf{n}}$ from the first column of $\mathbf{P}$. To recover $\mathbf{A}$, we also store the eigenvalues $\boldsymbol{\lambda}$.

As noted above, the canonical decomposition $\mathbf{A} = \mathbf{P}\boldsymbol{\Lambda}\mathbf{P}^\mathsf{T}$ does not necessarily lead to a matrix $\mathbf{P}$ whose first column equals the surface normal in sign. However, if we already know the (approximate) normal, which is the case in our simplification algorithm (see Section 4), then we can test whether the normal obtained from $\mathbf{P}$ matches the given normal. If the two vectors point in opposite directions, then we encode this fact by negating λ_1. Because $\mathbf{A}$ is non-zero and positive semi-definite, we must have $\lambda_1 > 0$, and we can therefore safely use the sign bit of λ_1 to encode the sign of $\hat{\mathbf{n}}$.

As mentioned in Section 4, we explicitly store approximate normals in the vertex files. To save disk space, however, we could use the technique just described for extracting normals from quadric matrices. We would then compute, but not store, a surface normal for each leaf node during simplification, compare it against the normal obtained from the quadric matrix $\mathbf{Q}$, and encode its sign difference in $\mathbf{Q}$. Similar to the argument above, because $\mathbf{Q}$ is non-zero and positive semi-definite, $\operatorname{tr}(\mathbf{Q}) > 0$, and the need to flip the extracted normal can be encoded by negating the diagonal of $\mathbf{Q}$.



(a) Orthogonal view. (b) Oblique view.

Figure 1: Illustration of silhouette preservation. The nearly flat faces of the cube can be simplified greatly when they are orthogonal to the view direction.

6.1.3 Curvature Matrix

Our view-dependent error metric takes advantage of the fact that geometric displacements parallel to the view direction are less perceptible than those orthogonal to the view direction. Thus geometry viewed straight-on can often be coarsened significantly more than geometry near silhouettes—a fact that has been exploited by other view-dependent methods, e.g. [17, 22, 25, 33]. This is illustrated in Figure 1 for a smoothed and slightly curved cube. Rather than using a cone to bound the normals [17, 25], we account for this directionality by analyzing the normal spread given by the quadric matrix.

Let $\hat{\mathbf{n}}_t$ be the unit normal, and consequently the direction of geometric error, associated with triangle t. Furthermore, let $\hat{\mathbf{v}}$ be the

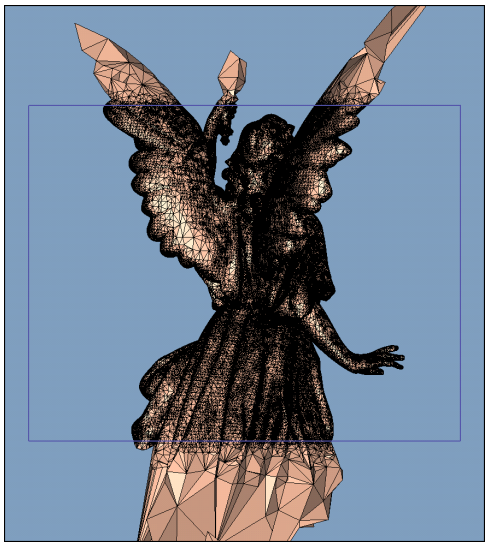


Figure 2: Illustration of culling against the view frustum (here shown as a rectangle).

unit vector from the cluster's representative vertex that we are computing the error for to the viewpoint. In our anisotropic error projection, we modulate the error associated with t by the sine of the angle γ_t between $\hat{\mathbf{n}}_t$ and $\hat{\mathbf{v}}$, i.e. by the factor $\eta_t = |\sin\gamma_t| = \|\hat{\mathbf{n}}_t \times \hat{\mathbf{v}}\|$. Thus, when γ_t is zero the projected error vanishes, while $\gamma_t = 90^\circ$ implies that we are near a silhouette and the projected error is at a maximum. We can rewrite (the square of) η_t as follows:

$$\eta_t^2 = \|\hat{\mathbf{n}}_t \times \hat{\mathbf{v}}\|^2 = \hat{\mathbf{v}}^\mathsf{T}\hat{\mathbf{v}}\hat{\mathbf{n}}_t^\mathsf{T}\hat{\mathbf{n}}_t - (\hat{\mathbf{v}}^\mathsf{T}\hat{\mathbf{n}}_t)^2 = \hat{\mathbf{v}}^\mathsf{T}(\mathbf{I} - \hat{\mathbf{n}}_t\hat{\mathbf{n}}_t^\mathsf{T})\hat{\mathbf{v}} \quad (13)$$

This modulation factor for a single triangle t can then be extended to a set of triangles T in a cluster as a weighted sum:

$$\begin{aligned}
\eta_T^2 &= \frac{\sum_{t\in T} w_t^2 \eta_t^2}{\sum_{t\in T} w_t^2} \\
&= \hat{\mathbf{v}}^\mathsf{T} \frac{\sum_{t\in T} w_t^2 \mathbf{I} - (w_t\hat{\mathbf{n}}_t)(w_t\hat{\mathbf{n}}_t)^\mathsf{T}}{\sum_{t\in T} w_t^2} \hat{\mathbf{v}} \\
&= \hat{\mathbf{v}}^\mathsf{T} \left(\mathbf{I} - \frac{\mathbf{A}}{\mathrm{tr}(\mathbf{A})} \right) \hat{\mathbf{v}} \\
&= \hat{\mathbf{v}}^\mathsf{T} \left(\mathbf{I} - \frac{\mathbf{P}\mathbf{\Lambda}\mathbf{P}^\mathsf{T}}{\lambda_1 + \lambda_2 + \lambda_3} \right) \hat{\mathbf{v}} \\
&= \hat{\mathbf{v}}^\mathsf{T}\mathbf{K}\hat{\mathbf{v}}
\end{aligned} \quad (14)$$

As in Section 4, w_t is the weight for t used in the quadric matrix construction; in our case, the area of t. The expression for the trace of $\mathbf{A}$, $\mathrm{tr}(\mathbf{A}) = \sum_{t\in T} w_t^2$, follows from the fact that $\mathrm{tr}(\cdot)$ is linear and $\mathrm{tr}(\hat{\mathbf{n}}_t\hat{\mathbf{n}}_t^\mathsf{T}) = \hat{\mathbf{n}}_t^\mathsf{T}\hat{\mathbf{n}}_t = 1$.

We call $\mathbf{K}$ the "curvature matrix," because it encodes the amount that the surface curves in different directions. From the definition of $\mathbf{K}$, we see that as the normals spread out the curvature grows, the eigenvalues approach a common value, $\mathbf{K}$ approaches $\frac{2}{3}\mathbf{I}$, and as a result η_T becomes more and more isotropic. If on the other hand λ_1 is much larger than the other eigenvalues, then the normals are tightly clustered, the curvature is small, and η_T allows aggressive coarsening when $\hat{\mathbf{v}}$ is near the dominant normal direction.

6.1.4 Bounding Sphere Radius

In order to support fast hierarchical view frustum culling, we compute a hierarchy of bounding spheres over the vertices in H. As in [23], the radius for a node p is given by

$$r_p = \begin{cases} 0 & \text{if } p \text{ is a leaf node} \\ \max\limits_{c\in C_p} \{\|\mathbf{p}_p - \mathbf{p}_c\| + r_c\} & \text{otherwise} \end{cases} \quad (15)$$

where C_p are the children of p. Using these bounding spheres for view frustum culling, we are guaranteed that a vertex is culled only if it and its descendants are not visible. Note that for conservative view culling the bounding sphere should also enclose all edge connected neighbors of a vertex [17]. However, because the adjacent vertices of a node depend nontrivially on run-time level-of-detail decisions, we cannot compute useful conservative bounding spheres off-line. In practice, our bounding sphere hierarchy is loose enough to not cause any visible artifacts. See Figure 2 for an example of view frustum culling.

in-core octree node		
vertex data		
3×3-matrix	$\mathbf{K}$	curvature matrix
3-vector	$\hat{\mathbf{n}}$	vertex normal
3-vector	$\mathbf{p}$	vertex position
scalar	ϵ	quadric error at $\mathbf{p}$
scalar	r	bounding sphere radius
triangle data		
count	n_T	number of triangles
vertex*3	T	list of triangles
count	n_R	number of references
pointer	R	list of references to this node
octree data		
boolean	*leaf*	is node a leaf on disk in H?
boolean	*front*	is node a leaf in memory in H'?
index	l	octree level
pointer	s	pointer to static external node
pointer*8	C	pointers to children
pointer	*prev*	previous node in queue
pointer	*next*	next node in queue

Table 4: In-core data structures for octree node.

7 IN-CORE DATA STRUCTURES

In this section we describe the in-core data structures used in our run-time view-dependent renderer. These data structures closely resemble the ones used by Luebke and Erikson [25], but have been modified to work in an out-of-core setting, where only part of the octree is assumed to be memory resident. That is, we maintain an in-core subset H' of the LOD hierarchy H, such that the leaf nodes of H' correspond to the vertices of the adaptively refined mesh. We refer to the subset H' of H as the set of *active* nodes. The triangles of the refined mesh are those stored in the internal (non-leaf) nodes of H'. Each triangle vertex, represented on disk as an octcode v for a leaf node in H, is mapped to a *proxy* vertex—either v itself or its lowest active ancestor. Thus, the actual vertices used for a triangle change dynamically as the mesh is adaptively refined and simplified.

Table 4 lists the octree node data structures that are dynamically allocated at run-time. We have already covered the fields for the vertex data in Section 6, and will here discuss the remaining fields for the node. The triangle data consists of a list of triangles and a list of pointers to triangle vertices that currently reference this node as their proxy. Thus the **vertex** data type consists of an octcode for a leaf node and an in-core octree node pointer to the vertex's proxy. Whenever a node is expanded, we need to modify all the triangle vertices that have the node as a proxy, which is accomplished by maintaining a list of back references to those vertices. Similarly, when a node is collapsed, its children's references are first accessed, and the associated proxies are modified to point to the collapsed node.

In contrast to [25], where the entire hierarchy is assumed to be memory resident, we cannot pre-compute the list of back references (or "tris" using their terminology), because these references might point to triangles in nodes that are not active and therefore have not been dynamically allocated. We could of course store these references as non-memory-specific octcodes, but have opted instead to maintain only references to triangles in active nodes, and to update the reference lists on-the-fly as triangles are added and removed. Similarly, we do not construct the triangle list T for a node until it is expanded. We will further discuss the operations on the octree H' in the following section.

The octree node pointers $prev$ and $next$ are part of a doubly linked list of all active nodes in H', except for leaf nodes in H, which cannot be expanded and for which collapsing has no effect. This list is traversed linearly during run-time refinement, as explained in the next section.

8 VIEW-DEPENDENT REFINEMENT

We are now ready to describe the steps pertaining to the final phase of our out-of-core view-dependent renderer: the run-time component. Because the level-of-detail hierarchy stored on disk may exceed the amount of available memory, we must be careful to page in only the active nodes of the hierarchy. Rather than making use of an explicit paging system, we rely on the use of read-only *memory mapping* to associate the on-disk hierarchy with a logically contiguous address space, and let the operating system fetch the data from disk when it is first accessed. Similar strategies for out-of-core rendering of large terrain have been employed, for example by Hoppe [18] and by Lindstrom and Pascucci [23]. This approach to data paging is particularly attractive when the refinement and rendering tasks are decoupled and run asynchronously, as is the case in our system. Also, as demonstrated in [23], by arranging the data and the accesses to it in a cache coherent manner, it is possible to substantially improve the paging performance. Indeed, our choice of arranging the octree in a coarse-to-fine, breadth-first layout on disk was made intentionally after having been inspired by the quadtree layout in [23]. We point out that since we do not access the memory mapped hierarchy continuously, but copy the on-disk data to an in-core structure when a node becomes active, it is entirely possible to use an explicit paging scheme in place of memory mapping. The benefits of doing so include improved scalability (many operating systems limit memory mapping to a 32-bit addressing space), more general geometry cache replacement policies (although the `madvise` system call, when implemented, provides this functionality), as well as potentially better memory usage if only small pieces of each memory page is needed (which is generally not the case in our system). The downsides of explicit paging include higher paging overhead, the need to keep an explicit "page table" for resident data, and implementing a robust data replacement policy that adapts well to the dynamically changing memory resources in a multiuser/multiprocess environment.

Given the general framework above, we now describe the two tasks of adaptive refinement and rendering.

8.1 Refinement Algorithm

Similar to several other components of our algorithm, our adaptive octree refinement closely follows the strategy employed by Luebke and Erikson [25]. We begin by creating a single node for the root of the dynamic octree H'. During refinement, we make use of two complementary operations; node expansion and collapse. When expanding a node we add its children; when collapsing a node we remove its descendants. Pseudo-code for these steps is listed in Table 5. Note that node expansion is valid only if a node is on the *front*, i.e. if it is a leaf node in H'.

```
node-expand(p)
 1  front(p) ← false
 2  for each child c of p
 3      initialize c by computing its vertex data
 4      if ¬leaf(c) then
 5          append c to end of refinement queue Q
 6  for each triangle t assigned to p
 7      for each vertex v_t^i of t
 8          identify proxy node q_t^i from octcode v_t^i
 9          add reference r from proxy node q_t^i to t
10  for each reference r in p
11      identify child c corresponding to r
12      transfer r from p to c and update proxy pointer for r

node-collapse(p)
 1  for each child c of p
 2      node-collapse(c)
 3      for each reference r in c
 4          transfer r from c to p and update proxy pointer for r
 5      if ¬leaf(c) then
 6          remove c from refinement queue Q
 7      remove c from H'
 8  for each triangle t assigned to p
 9      for each vertex v_t^i of t
10          remove reference to t from proxy q_t^i for v_t^i
11  front(p) ← true
```

Table 5: Pseudo-code for node expansion and collapse.

As already mentioned, the refinement runs as a separate thread and produces a list of triangles to render by a dedicated render thread using a double buffering approach; one buffer is being displayed while the other is being worked on by the refinement thread. Rather than visiting the active nodes in a depth-first manner during refinement, we make use of a linear circular queue Q that orders the nodes roughly in a breadth-first manner. The reason for doing so is that we want the ability to preempt the refinement whenever large amounts of data need to be paged in, which allows the mesh to be incrementally updated periodically and progressively. The breadth-first traversal ensures that detail is paged in and added evenly over the visible surface. By specifying a minimum update frequency (we use a default of one update per second), the refinement thread can be preempted at any point, and we maintain a pointer into the circular queue where we last left off, in case the refinement was previously interrupted, or at the "tail" of the queue, if all nodes were previously processed. At the beginning of each refinement pass, we set the tail to the node last processed, and always add new nodes resulting from node expansion at the tail end of the queue. Note that this results in an approximate breadth-first ordering of the active nodes.

For each node visited, we test its bounding sphere against the six planes of the view frustum to determine if it is visible. If not, we collapse it. Thus pieces of the mesh outside the view frustum are not entirely discarded, but are aggressively coarsened. This allows at least crude navigation of the mesh whenever the view shifts rapidly. If the node is visible, we project its quadric error onto the screen. (The details of this evaluation are given below.) If the error exceeds a user-specified threshold τ and the node is on the front, we expand it. Otherwise, if the threshold is not exceeded and the node is not on the front, we collapse it. In this manner, the octree adapts as the viewpoint changes, and we visit only those nodes that eventually make up the mesh. Because nodes above the front are also evaluated, it is possible to quickly cull out large portions of the octree, e.g. if the view conditions change rapidly, without having to trim the octree bottom-up.

Finally, after the octree has been refined, we traverse it node by node and add all the triangles encountered to a triangle list. For each triangle, we follow the pointers to the proxy nodes and append

their vertex positions and normals to the list. If flat (per-triangle) shading is preferred, we instead compute and store triangle normals in the list. Upon completion, the triangle list is shipped to the render thread and gets reused until the next list has been constructed.

8.2 Screen Space Metric

The decision whether to collapse or expand a node is governed by a screen space error metric and a user-specified error threshold. In our screen space metric, we make use of the quadric error ϵ and position $\mathbf{p}$ of a node's representative vertex, as well as the curvature matrix $\mathbf{K}$. Note that, due to our triangle-area-weighting of the geometric displacements (see Section 4), our quadric error has units of volume squared, which needs to be expressed in units of length. This can be done, for example, by normalizing the error by dividing by the sum of squared areas for the cluster's triangles (i.e. by the sum of eigenvalues $\lambda_1 + \lambda_2 + \lambda_3$). Another approach, and the one used in our implementation, which ensures that the error of a node is at least as large as its children's, is to assume that the volumetric errors correspond to some hypothetical volume, e.g. a sphere, in which case we simply compute the radius of the sphere and use it as the error term. In either case, these computations are done once when the node is constructed, and the ϵ field in the dynamic node is assumed to have units of length.

As a base metric, we set the screen space error to be proportional to the ratio of the object space error and the distance from the viewpoint to the node. As suggested in Section 6.1.3, to incorporate directionality into our metric for silhouette preservation, we modulate the base metric by the factor η. Thus, our screen space metric ρ can be written as

$$\rho = \lambda\eta\frac{\epsilon}{\|\mathbf{e}-\mathbf{p}\|} = \lambda\epsilon\frac{\sqrt{\mathbf{v}^\mathsf{T}\mathbf{K}\mathbf{v}}}{\mathbf{v}^\mathsf{T}\mathbf{v}} \tag{16}$$

where $\mathbf{v} = \mathbf{e}-\mathbf{p}$ is the vector from the node to the viewpoint $\mathbf{e}$, and λ is the screen resolution in pixels per radians. We then compare ρ against a threshold τ to determine whether the node is active or not. For efficiency reasons, we square and rearrange some terms, and obtain the following expression:

$$\begin{aligned} active &\iff \rho > \tau \\ &\iff \lambda^2\epsilon^2(\mathbf{v}^\mathsf{T}\mathbf{K}\mathbf{v}) > \tau^2(\mathbf{v}^\mathsf{T}\mathbf{v})^2 \\ &\iff \epsilon^2(\mathbf{v}^\mathsf{T}\mathbf{K}\mathbf{v}) > \kappa^2(\mathbf{v}^\mathsf{T}\mathbf{v})^2 \end{aligned} \tag{17}$$

where $\kappa = \frac{\tau}{\lambda}$ is the screen space error threshold in radians.

Because the octree is pruned whenever a node is found to be inactive, we should ideally ensure that a node's projected error is always larger than those of its children. Note that the quadric errors by their additive nature already satisfy this nesting property. Similarly, the amount of curvature encoded in the matrix $\mathbf{K}$ can only increase when quadric matrices are combined. However, the direction and length of the vector $\mathbf{v}$ vary from node to node, making it possible to violate the nesting condition. A general technique for handling this view-dependent problem was presented in [23], in which a nested sphere hierarchy is computed and used in place of the positions of individual vertices. This sphere hierarchy is the same as the one we compute for view frustum culling purposes. While not implemented here, we believe that it would be rather straightforward to incorporate the sphere hierarchy into our screen space metric.

9 RESULTS

In this section we present experimental results of running our algorithms. We used a number of polygonal test models, including a massive 373 million triangle model of Michelangelo's St. Matthew statue [19]. All models, except the Buddha, were simplified on one processor of a 250 MHz R10000 SGI Onyx2 with 40.5 GB of main memory. The Buddha model was simplified on a Linux PC with two 800 MHz Pentium III processors, 880 MB of RAM, and a GeForce3 graphics card. We used this PC for the accompanying video of the Lucy data set. To stress the out-of-core aspect of our system, we used a lower-end dual-processor Linux PC, with 256 MB of RAM and GeForce2 graphics, for the Isosurface video.

Animations We have included two videos to illustrate the run-time performance of our system. As is evident in the accompanying Lucy video, we obtain a throughput of roughly three million rendered triangles per second using a GeForce3 graphics card. Notice the ability to adapt the mesh to the view frustum, the shape of the surface, and the presence of silhouettes (see, for example, the neck of the statue). The Isosurface data set is a very dense and complex surface of the turbulent boundary between two mixing fluids. Because we used backface culling, it is possible to see through much of the back side of the surface (the majority of the video sequence). This video illustrates the progressive nature of the refinement, as mesh updates are made at least once per second, and the rather fast paging of data as the mesh is navigated at various scales. The slower rendering speed is due to the use of a lower performance graphics card, as well as memory contention due to the limited amount of memory available. Note that we were careful to invalidate the operating system's disk cache before creating these animations, to ensure that no data was memory-resident at startup. In addition to the videos, Figure 3 shows qualitative results of our run-time method.

Disk and Memory Usage Table 6 lists numerical results for our off-line method, including the number of triangles, grid size (2^n), timing results, effective triangle processing rate, and disk usage. Not included in this table is the maximum memory usage. Our method uses only a constant $O(1)$ amount of memory: 5 MB for the Linux machine and 8 MB for the SGI (the difference is due to different size executables). The temporary disk usage of our method can be shown to be a constant multiple (roughly a factor of 5) of the size of the input mesh. While the theoretical usage is linear in both the size of the input and the output, the simplification phase often reduces the mesh to the extent that the overall temporary disk space is entirely dominated by the plane file (which is removed before the hierarchy construction begins) and any disk space used while sorting this file. As can be seen from the last column, the size of the octree output file is on average only a few percent larger (in bytes per triangle) than the size of the input. The output ranges from 35–45 bytes per triangle. For the sake of fairness, our triangle soup input format is not as efficient as the more common indexed mesh representation, which requires only half as much storage. The indexed mesh, on the other hand, is impractical for external memory algorithms since it requires random access.

Execution Time As can be seen from the table, for small octrees the total time is dominated by the external sort phase of the simplification. As the octree (and thus size of the output) grows larger, the output phase of the octree construction becomes more significant, but generally requires only a small additional fraction of the simplification time. Even though the off-line construction is done entirely on disk, our memory insensitive algorithm is remarkably fast, and yields an effective triangle processing rate (measured as the size of the input over the total time) of roughly 20,000–60,000 triangles per second (tps). As a point of reference, El-Sana and Chiang [8] report a reduction rate of roughly 5,300 tps for their largest model, consisting of 1.2 million triangles. This model is considerably smaller than some of those simplified here, and the nature of their method suggests that it would steadily decline in performance as the input grows larger. The method proposed by Prince [27], while producing high quality meshes, yields about 1,000 tps for the largest model used (11.4 million triangles).

model	T_{in}	T_{out}	n	*simp. time (%)*			*hier. time (%)*			*total time*	T_{in}/s	*disk usage (MB)*		
				read	*sort*	*write*	*sort*	*pull*	*push*	*(h:m:s)*		*input*	*temp.*	*output*
Buddha	1,087,716	62,346	7	20.6	48.0	25.7	0.5	2.7	2.6	18	61,468	37	189	2.2
		204,766	8	17.2	40.7	25.6	1.3	7.9	7.3	22	49,622		189	7.7
		522,700	9	12.7	28.9	25.0	2.4	17.1	14.0	32	33,908		189	22.7
Lucy	28,055,742	188,782	8	28.9	37.7	32.0	0.1	0.9	0.4	11:46	39,737	963	4,713	6.5
		721,798	9	27.0	37.0	30.9	0.2	3.4	1.4	12:40	36,897		4,823	25.5
		2,678,781	10	27.6	35.4	23.9	0.5	9.0	3.5	18:37	25,119		4,815	96.8
Isosurface	228,996,372	46,949,908	10	21.1	25.5	27.4	1.5	17.5	7.0	2:39:10	23,978	7,862	39,994	1,585.2
St. Matthew	372,767,445	3,187,812	10	27.9	42.2	28.3	0.1	1.1	0.5	2:51:40	36,190	12,798	63,942	112.6

Table 6: Numerical results for memory insensitive simplification and level-of-detail hierarchy construction. The timings are reported for the subphases *read* (lines 1–8), *sort* (line 9), and *write* (lines 10–15) of the simplification phase (see Table 1), and *sort* (line 1), *pull* (lines 2–9) and *push* (lines 10–17) of the hierarchy construction phase (see Table 2). The Buddha data set was simplified on a Linux PC, while the other models were simplified on an SGI Onyx2.

Meanwhile, his method requires more than 512 MB of RAM to simplify this model, whereas we use a constant 8 MB. Closest in performance to our method is the one by Cignoni et al. [4], with up to 13 thousand tps.[2] However, the output of their method is not in a form that is directly usable for view-dependent refinement.

The theoretical execution time of our off-line processing is $O(T_{in}+T_{out})$, assuming the external sort is implemented as a radix sort,[3] which suggests that our method scales well.

10 SUMMARY AND FUTURE WORK

We have described a method for constructing a level-of-detail hierarchy for large polygonal meshes. This method performs all computations on disk, and uses only a small, constant amount of RAM. The method requires temporary disk space linear in the size of the input and output, and runs in linear time. Even though virtually no RAM is used, our method executes one to two orders of magnitude faster than previous methods, and achieves a peak simplification rate of over 60,000 triangles per second. We have also presented compact data structures and a suitable error metric for performing out-of-core view-dependent refinement of the resulting mesh hierarchy. Our results show that we obtain interactive frame rates and a throughput of around three million triangles per second using immediate mode rendering.

We see several avenues for future work. One limitation of our system is that the multiresolution surface constructed does not retain the fidelity of the original mesh. One possible solution to this problem is to keep the triangles that degenerate during simplification and store them verbatim with their corresponding leaf nodes. Another limitation is that our 32-bit octcodes support a maximum octree depth of 10. Using 64-bit codes would allow a finer grid and less coarsening of the original data set, although at the expense of conciseness of representation. Our run-time system could be improved via the use of geomorphing to smooth out temporal popping artifacts, and prefetching to improve the performance of the paging system. We believe that additional prefetch threads that visit nodes below the active front would improve the paging performance. We also intend to investigate how to extend our error metric to guarantee the nesting condition in screen space, and how to account for the visual impact of simplification on the shading of the surface.

Finally, we envision that our off-line algorithms can be further enhanced. As noted in [24], the greatest potential for reducing the disk usage lies in using a compressed, perhaps quantized representation for the rather large plane file, which by far dominates the amount of temporary disk space used. We also see untapped potential in increasing the speed of our out-of-core method through parallelization. The locality of our data accesses and the octree partitioning of space and work naturally lend themselves to parallel execution.

[2]This does not include the time required to first construct their external memory representation of the original mesh. When included, the overall performance drops to around 6 thousand tps.

[3]The external sort `rsort` used in our implementation is a combination of radix and merge sort.

Acknowledgements

We would like to thank Stanford University and the Digital Michelangelo Project for providing the Buddha, Lucy, and St. Matthew data sets.

References

[1] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An Interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration. *1999 ACM Symposium on Interactive 3D Graphics*, 199–206. Apr. 1999.

[2] F. Bernardini, J. Mittleman, and H. Rushmeier. Case Study: Scanning Michelangelo's Florentine Pietà. SIGGRAPH 99 Course #8, Aug. 1999.

[3] P. Choudhury and B. Watson. Completely Adaptive Simplification of Massive Meshes. *Tech. Rep. CS-02-09*, Northwestern University, Mar. 2002. URL http://www.cs.northwestern.edu/~watsonb/school/docs/vmrsimp.tr.pdf.

[4] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External Memory Management and Simplification of Huge Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 2002. To appear.

[5] J. Cohen, D. G. Aliaga, and W. Zhang. Hybrid Simplification: Combining Multi-Resolution Polygon and Point Rendering. *IEEE Visualization 2001*, 37–44. Oct. 2001.

[6] C. DeCoro and R. Pajarola. XFastMesh: Fast View-Dependent Meshing from External Memory. *IEEE Visualization 2002*, 363–370. Oct. 2002.

[7] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. *IEEE Visualization '97*, 81–88. Nov. 1997.

[8] J. El-Sana and Y.-J. Chiang. External Memory View-Dependent Simplification. *Computer Graphics Forum*, 19(3):139–150, Aug. 2000.

[9] J. El-Sana and A. Varshney. Generalized View-Dependent Simplification. *Computer Graphics Forum*, 18(3):83–94, Sep. 1999.

[10] C. Erikson, D. Manocha, and W. V. Baxter III. HLODs for Faster Display of Large Static and Dynamic Environments. *2001 ACM Symposium on Interactive 3D Graphics*, 111–120. Mar. 2001.

[11] G. Fei, K. Cai, B. Guo, and E. Wu. An Adaptive Sampling Scheme for Out-of-Core Simplification. *Computer Graphics Forum*, 21(2):111–119, Jun. 2002.

[12] T. A. Funkhouser and C. H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Proceedings of SIGGRAPH 93*, 247–254. Aug. 1993.

[13] M. Garland. *Quadric-Based Polygonal Surface Simplification*. Ph.D. thesis, Carnegie Mellon University, May 1999.

[14] M. Garland and P. S. Heckbert. Surface Simplification Using Quadric Error Metrics. *Proceedings of SIGGRAPH 97*, 209–216. Aug. 1997.

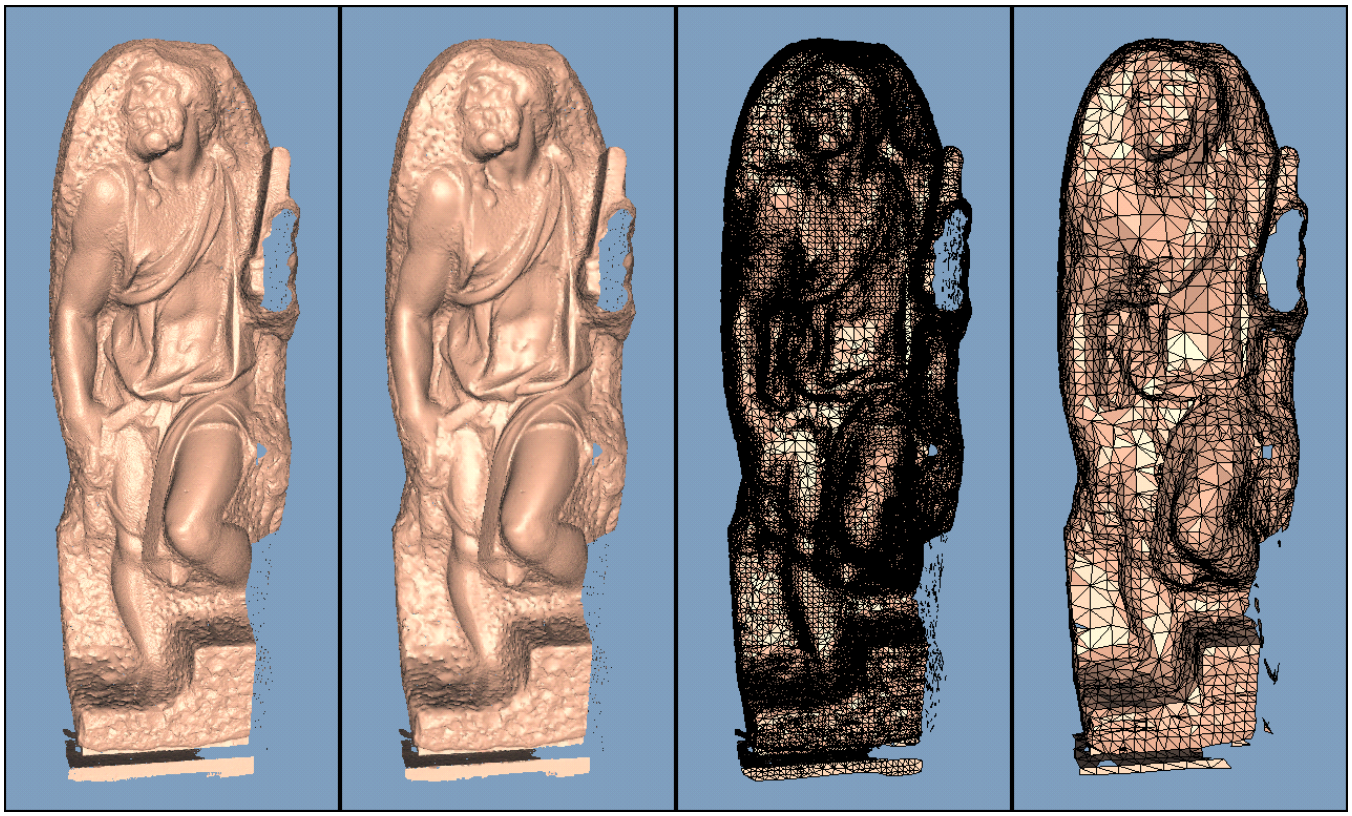
[15] M. Garland and E. Shaffer. A Multiphase Approach to Efficient Surface Simplification. *IEEE Visualization 2002*, 117–124. Oct. 2002.

[16] H. Hoppe. Progressive Meshes. *Proceedings of SIGGRAPH 96*, 99–108. Aug. 1996.

[17] H. Hoppe. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH 97*, 189–198. Aug. 1997.

[18] H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. *IEEE Visualization '98*, 35–42. Oct. 1998.

[19] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. *Proceedings of SIGGRAPH 2000*, 131–144. Jul. 2000.



(a) 2,860,000 triangles. (b) 1,216,000 triangles. (c) 166,000 triangles. (d) 15,000 triangles.

Figure 3: View-dependent renderings of the St. Matthew data set for various error thresholds. The original model contains 373 million triangles.

[20] J. Linderman. `rsort` man page, Apr. 1996. Revised Jun. 2000.

[21] P. Lindstrom. Out-of-Core Simplification of Large Polygonal Models. *Proceedings of SIGGRAPH 2000*, 259–262. Jul. 2000.

[22] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner. Real-Time, Continuous Level of Detail Rendering of Height Fields. *Proceedings of SIGGRAPH 96*, 109–118. Aug. 1996.

[23] P. Lindstrom and V. Pascucci. Visualization of Large Terrains Made Easy. *IEEE Visualization 2001*, 363–370. Oct. 2001.

[24] P. Lindstrom and C. T. Silva. A Memory Insensitive Technique for Large Model Simplification. *IEEE Visualization 2001*, 121–126. Oct. 2001.

[25] D. Luebke and C. Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. *Proceedings of SIGGRAPH 97*, 199–208. Aug. 1997.

[26] D. Luebke and B. Hallen. Perceptually-Driven Simplification for Interactive Rendering. *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 223–234. Jun. 2001.

[27] C. Prince. *Progressive Meshes for Large Models of Arbitrary Topology*. Master's thesis, University of Washington, 2000.

[28] J. Rossignac and P. Borrel. Multi-Resolution 3D Approximations for Rendering Complex Scenes. *Modeling in Computer Graphics*, 455–465. Springer-Verlag, 1993.

[29] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. *Proceedings of SIGGRAPH 2000*, 343–352. Jul. 2000.

[30] E. Shaffer and M. Garland. Efficient Adaptive Simplification of Massive Meshes. *IEEE Visualization 2001*, 127–134. Oct. 2001.

[31] K. Shoemake. Quaternions and 4x4 Matrices. *Graphics Gems II*, 351–354. Academic Press, 1991.

[32] G. Varadhan and D. Manocha. Out-of-Core Rendering of Massive Geometric Environments. *IEEE Visualization 2002*, 69–76. Oct. 2002.

[33] J. C. Xia and A. Varshney. Dynamic View-Dependent Simplification for Polygonal Models. *IEEE Visualization '96*, 327–334. Oct. 1996.